Étape 5 : Animaux et premiers déplacements

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.1

But

Continuer à modéliser les animaux et leur permettre de se déplacer de façon simple.

Description

Dans cette étape, nous poursuivons la modélisation des points communs entre les animaux (fourmis et termites) au moyen de la classe abstraite `Animal`. Une méthode de déplacement simple, potentiellement utilisable par tous les animaux, sera introduite dans cette classe. Les animaux seront dotés d'une durée de vie limitée et disparaîtront de la simulation une fois leur limite d'âge atteinte.

La classe `Environment` sera retouchée de sorte à permettre la simulation des animaux et leur rendu graphique. Les termites seront également un peu élaborées de sorte à ce que nous puissions visualiser un premier exemple d'animaux de déplaçant et mourant.

Pour que l'impact que peuvent avoir les animaux sur l'environnement ne sorte pas de leurs prérogatives naturelles (on ne veut pas par exemple qu'un animal génère une source de nourriture dans l'environnement), on introduira une interface `AnimalEnvironmentView` analogue à l'interface `FoodEnvironementView` introduite précédemment pour les `Food`.

La figure 1 donne une vue d'ensemble de l'architecture à laquelle vous allez aboutir au terme de cette étape.

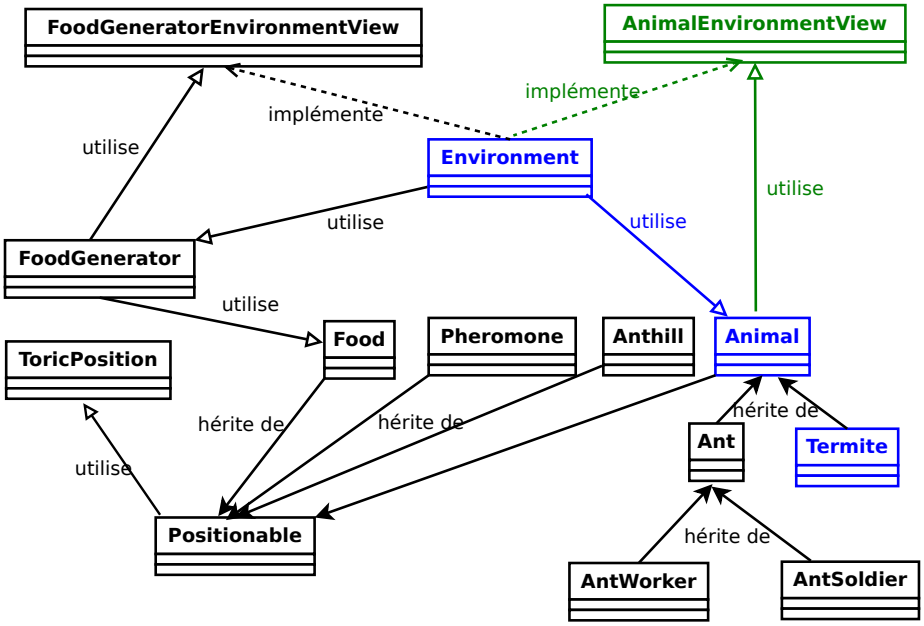

Figure 1: Architecture codée au terme de l'étape 5 (en noir les composants déjà codés, en vert les nouveaux composants, en bleu ceux retouchés)

Implémentation

Classe `Animal`

Reprenez la classe abstraite `Animal` du package `ch.epfl.moocprog`.

Complétez la modélisation de cette classe en la dotant :

- d'une direction de déplacement (voir plus bas) ;
- d'un nombre de points de vie représenté par un `int` (cela servira plus tard à mesurer les dégâts subis lors d'affrontement avec des congénères et estimer les chances de survie) ;
- d'une durée de vie maximale (de type `Time`).

Note : pour les besoins de la correction automatique, nous devons exceptionnellement vous imposer des noms particulier pour les attributs de la classe `Animal`. Vous devez utilisez le nom `hitpoints` pour les points de vie et le nom `lifespan` pour la durée maximale de vie. La classe `Animal`sera la seule à vous contraindre de la sorte dans le nommage des attributs.

Direction de déplacement

La direction de déplacement peut être caractérisée par l'angle que fait le vecteur direction de l'animal (un vecteur l'orientant droit devant lui) par rapport à l'axe des x dans ce repère ; nous appellerons cet angle, *l'angle de direction* (voir figure 2).

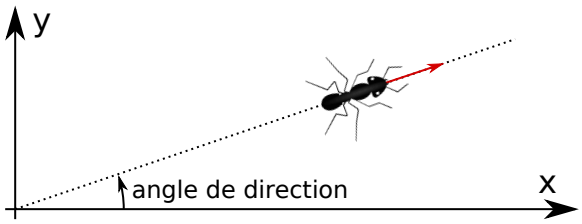

Figure 2: angle de direction des animaux.

Cet angle est mesuré en radians car les fonctions trigonométriques utilisent cette unité en Java.

Vous retoucherez la méthode `getDirection` introduite à l'étape précédente pour lui faire retourner la valeur de l'attribut modélisant la direction.

Méthodes et constructeur

Vous doterez ensuite votre classe des méthodes suivantes :

- `toString()` retournant une représentation textuelle d'un `Animal` selon le format de l'exemple suivant :

  ```
  Position : 20.0, 30.0
  Speed : 120.0
  HitPoints : 500
  LifeSpan : 30.000000
  ```

Les libellés `Position`, `Speed`, `HitPoints` et `LifeSpan` sont importants pour une bonne intégration avec l'interface graphique à venir. Vous veillerez à réutiliser proprement les méthodes `toString` héritées de plus haut.

- `void setDirection(double angle)` modifiant la valeur de l'angle de direction ;
- `boolean isDead()` retournant `true` si le nombre de points de vie de l'animal ou sa durée de vie est inférieur(e) ou égal(e) à zéro.

Ces deux méthodes ne devraient pas pouvoir être redéfinies par les classes filles.

Vous modifierez le constructeur :

- il initialisera la position, le nombre de points de vie et la durée de vie avec des valeurs fournies en paramètres (dans cet ordre). L'angle de direction sera initialisé de façon aléatoire entre `0` et `2 * Math.PI`, de manière uniforme. Vous pouvez vous aider de la méthode statique

```
double getValue(double min, double max)
```

de la classe `UniformDistribution` située dans le package `ch.epfl.moocprog.random`.

Vous ajouterez également les méthodes `getHitpoints()` et `getLifespan()` retournant simplement l'attribut correspondant.

Par ailleurs, tout `Animal` aura aussi une vitesse de déplacement. Nous considérons cependant que la vitesse peut potentiellement se calculer et ne peut pas être définie pour un animal abstrait. Nous faisons donc le choix de ne pas introduire d'attribut pour la vitesse mais plutôt une méthode abstraite `double getSpeed()` qui devra être redéfinie de façon adéquate dans les sous-classes.

Les termites comme premières instances concrètes d'animaux

Pour l'heure, les animaux que nous avons introduits à l'étape précédente restent abstraits car héritant de `getSpeed`. Nous allons utiliser les termites comme premières classes d'animaux concrets que nous pourrons visualiser en train de se déplacer. Pour cela dotez la classe `Termite` d'une définition concrète de la méthode `getSpeed`. Cette dernière retournera simplement la valeur du paramètre :

```
getConfig().getDouble(TERMITE_SPEED)
```

La vitesse de déplacement des termites devient ainsi paramétrable depuis l'extérieur et sa valeur pourra ainsi être changée à loisir, sans même avoir besoin de relancer le programme !

Retouchez également le constructeur de la classe `Termite` de sorte à ce qu'il initialise les points de vie et la durée de vie au moyen de valeurs paramétrable, concrètement :

```
getConfig().getInt(TERMITE_HP)
```

et

```
getConfig().getTime(TERMITE_LIFESPAN)
```

Nous délaissons délibérément la modélisation des fourmis pour le moment. Dotez simplement les classes `AntWorker` et `AntSoldier` d'une méthode `getSpeed` retournant zéro (afin de ne pas avoir à déclarer ces classes comme abstraites, ce qui ne fait pas sens !)

Méthode de simulation et vue sur l'environnement

Nous allons maintenant mettre en place la méthode qui va servir à faire évoluer un animal au cours du temps. Comme les animaux auront besoin d'interagir avec les autres entités de l'environnement, nous aurons besoin d'avoir une vue sur ce dernier (revoir si nécessaire le sujet de l'étape 3). Créez donc l'interface `AnimalEnvironmentView` et laissez-la vide pour le moment. Nous viendrons la compléter plus tard. De façon analogue à ce que nous avons fait pour les `Food`, cette interface contiendra les méthodes qu'un `Animal` est en droit d'utiliser pour agir sur l'environnement. Faites en sorte que `Environment` implémente cette nouvelle interface.

Il est temps maintenant de rendre nos animaux un peu plus « dynamique » ! Ajoutez pour cela à la classe `Animal` la méthode

```
void update(AnimalEnvironmentView env, Time dt)
```

qui sera en charge de faire évoluer un animal sur « un pas de temps » `dt` (cette méthode simule ce qui change pour l'animal lorsqu'un intervalle de temps `dt` s'écoule). Pour l'instant, le temps n'aura d'impact que sur sa durée de vie. La méthode `update` se contentera de soustraire à la durée de vie le temps `dt` multiplié par `ANIMAL_LIFESPAN_DECREASE_FACTOR` (qui est un `double`).

Pour obtenir la valeur de `ANIMAL_LIFESPAN_DECREASE_FACTOR`, utilisez la méthode `getConfig()` :

```
getConfig().getDouble(ANIMAL_LIFESPAN_DECREASE_FACTOR)
```

Pour effectuer les opérations décrites sur les objets `Time`, utilisez les méthodes `minus()` (soustraction) et `times()` (multiplication).

Nous n'utilisons pas pour le moment le premier paramètre (de type `AnimalEnvironmentView`), mais ce sera le cas dans les étapes à venir.

Ce paramètre signifie qu'en évoluant, un animal pourra agir sur la vue qu'il a de l'environnement ; cette vue le restreignant dans ce qu'il a le droit de faire !

Déplacement rectiligne uniforme

Le modèle de déplacement des animaux sera très simple : on augmente la position de *dt* fois la vitesse, où *dt* est le « pas de temps » utilisé. La formule sera un tout petit peu plus compliquée car on souhaite que l'animal garde sa direction (angle de direction décrit plus haut).

Dotez la classe `Animal` d'une méthode

```
void move(Time dt)
```

qui met à jour la position de l'animal au bout de l'écoulement du « pas de temps » `dt`. Pour cela, commencez par calculer le `Vec2d` correspondant au déplacement :

c'est simplement le vecteur donnée par l'angle de direction de l'animal (méthode `fromAngle()` de `Vec2d`) multiplié (méthode `scalarProduct()` de `Vec2d`) par le produit de `dt` et la vitesse de l'animal. Le temps `dt` étant de type `Time`, il doit être converti en une valeur numérique pour effectuer les calculs. Vous utiliserez pour cela la méthode `toSeconds()`de la classe `Time` (fournie dans le dossier `utils`). Utilisez ensuite la méthode `add()` des `ToricPosition` pour ajouter ce déplacement à la position courant de l'animal.

La méthode `move()` devra être `protected` et ne devra pas être redéfinissable par les sous-classes. Elle ne devra être appelée par la méthode

```
void update(AnimalEnvironmentView env, Time dt)
```

mais uniquement si l'animal est encore en vie. Ce comportement sera modifié par la suite : il existe des situations (autre que le décès) où l'animal ne devra pas être continuellement en mouvement (par exemple lors des combats, l'animal ne devra plus bouger). Nous y reviendrons au moment voulu.

Déambulations

Nous allons maintenant doter la classe `Environment` d'un attribut de type `List<Animal>` : cela permettra de répertorier tous les animaux qui ont part à la simulation. Définissez également la méthode publique

```
void addAnimal(Animal animal)
```

qui permet, comme son nom le suggère, d'ajouter des animaux à l'environnement. Comme pour les `Food`, la méthode `addAnimal` devra lancer une `IllegalArgumentException` si le paramètre `animal` vaut `null`.

Comme pour le cas de `Food`, le type dynamique de cette liste devra être une `LinkedList<Animal>` car la collection d'animaux doit pouvoir supporter des suppressions efficaces en cours d'itération.
Rajoutez dans la méthode `update` de `Environment` le parcours et la suppressions des animaux : pour chaque animal dans la liste, il faudra tester s'il est mort. Si c'est le cas, il faudra le retirer de la liste. Sinon, on appellera sa méthode de mise à jour `update()` (celle d'`Animal`).

L'ordre des méthodes de mises à jour est le suivant :

- génération de nourriture ;
- gestion des animaux ;
- nettoyage des instances de nourriture avec une quantité nulle.

Ajoutez également la méthode publique `List<ToricPosition> getAnimalsPosition()` dans `Environment`. Cette méthode devra simplement renvoyer la liste des positions des animaux.

Indication : Pour supprimer un élément d'une liste sur laquelle l'on est en train d'itérer, il faut utiliser la notion d'itérateur selon le schéma suivant :

```
Iterator<UneClasse> iterateur = collectionDeUneClasse.iterator();
while(iterateur.hasNext()) {
    UneClasse instanceDeUneClasse = iterateur.next();
    if(/* une condition sur instanceDeUneClasse */) {
      iterateur.remove();
```

sans passer par les itérateurs, vous vous exposez en effet au risque d'occurence d'une `ConcurrentModificationException`. Pour plus de détails à ce sujet, voir le complément de cours (document PDF) sur le site du MOOC à l'étape correspondante (étape 05).

Pour finir, modifiez la méthode `renderEntities` de Environment de sorte à ce que les animaux simulés soient aussi dessinés (procédez de façon analogue à ce que vous avez fait pour la collection de nourriture).

Tests et soumission

Tests locaux

Nous vous encourageons, comme toujours à continuer d'ajouter vos propres tests ponctuels dans le fichier `ch/epfl/moocprog/tests/Main.java`.

Les termites sont à ce stade des animaux basiques qui héritent de la méthode de déplacement de `Animal` et qui sont dotés d'une durée de vie limitée. Elles nous permettent donc déjà de tester et visualiser les éléments codés dans cette étape.

Voici un exemple minimal de code que vous pourriez ajouter au fichier `ch/epfl/moocprog/tests/Main.java` pour réaliser quelques tests simples :

```
// Quelques tests pour l'étape 5
System.out.println();
System.out.println ("A termite before update :");
Termite t1 = new Termite(new ToricPosition(20, 30));
System.out.println(t1);
env.addAnimal(t1);
env.update(Time.fromSeconds(1.));
System.out.println("The same termite after one update :");
System.out.println(t1);
```

Ce qui devrait produire un affichage comme celui-ci :

```
A termite before update :
Position : 20.0, 30.0
Speed : 120.0
HitPoints : 500
```

```
LifeSpan : 30.000000

The same termite after one update :
Position : 910.9, 680.0
Speed : 120.0
HitPoints : 500
LifeSpan : 29.700000
```

Vous êtes évidemment aussi encouragés à utiliser l'interface graphique pour tester globalement le comportement de tout le système.

Le fichier de configuration `res/config.cfg` prévoit la création de 7 termites (toutes situées au même endroit) au lancement du programme.

Figure 3: Les termites s'élancent droit devant elles et se déplacent selon un mouvement rectiligne uniforme

Lancez le programme graphique comme indiqué dans l'étape précédente. Vous devriez voir s'afficher dans le coin en haut à gauche, une amas de termites les unes sur les autres. Si le déplacement a été correctement codé, en appuyant sur le bouton `Play` vous devriez les voir s'élancer droit devant elle (normalement avec des directoins de départ différentes). La figure 3 vous donne un exemple (figé) de ce que vous devriez observer.

Vérifiez :

- que la durée de vie des termites décroît continuellement (au moyen du bouton `LifeSpan`) ;
- que leur vitesse est constante et proprement initialisée (au moyen du bouton `Speed`) ;

- que la modification de leur vitesse dans le panneau de droite a bien un impact (elles doivent aller plus ou moins vite selon la valeur donnée à `TERMITE_SPEED`).

Vous pouvez accélérer la simulation (`TIME_FACTOR` à 5 par exemple) pour constater plus rapidement que les termites meurent bel et bien lorsque la durée de vie atteint la valeur nulle.

N'oubliez pas de faire un `Reset` avant chaque `Restart` ou avant de quitter le programme de simulation si vous voulez conserver les valeurs d'origine du fichier `res/app.cfg`.

Une petite vidéo d'exemple de simulation obtenue à ce stade est disponible dans le matériel de la semaine 3.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **tous** vos fichiers sources personnels, depuis la racine `ch/` ; ceux de cette étape, mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

```
ch/epfl/moocprog/AnimalEnvironmentView.java
ch/epfl/moocprog/Animal.java
ch/epfl/moocprog/Anthill.java
ch/epfl/moocprog/Ant.java
ch/epfl/moocprog/AntSoldier.java
ch/epfl/moocprog/AntWorker.java
ch/epfl/moocprog/Environment.java
ch/epfl/moocprog/FoodGeneratorEnvironmentView.java
ch/epfl/moocprog/FoodGenerator.java
ch/epfl/moocprog/Food.java
ch/epfl/moocprog/Pheromone.java
ch/epfl/moocprog/Positionable.java
ch/epfl/moocprog/Termite.java
ch/epfl/moocprog/ToricPosition.java
```

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). L'énoncé « *Procédure de soumission* » de la semaine 1 vous indique comment procéder. **Avant de passer à l'étape suivante, n'oubliez pas de faire une sauvegarde de l'étape en cours**, comme indiqué dans le même document.